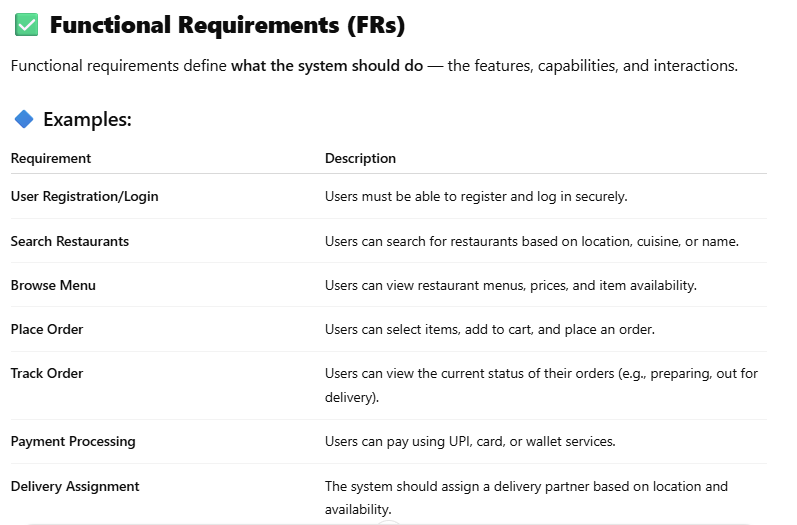
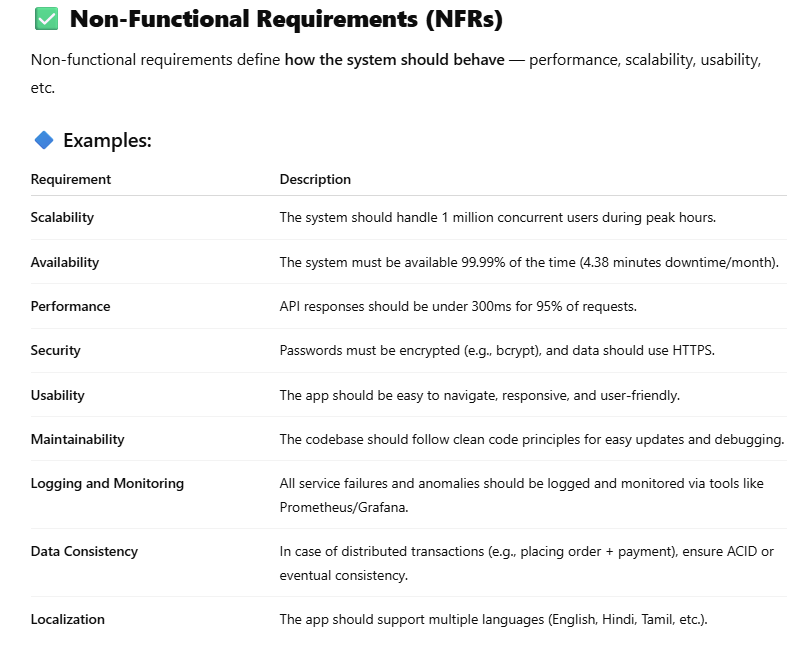
**Functional and Non Functional Requirements:**

Let's break down the functional and non-functional requirements using a real-world system example: an Online Food Delivery Platform (like Swiggy or Zomato).

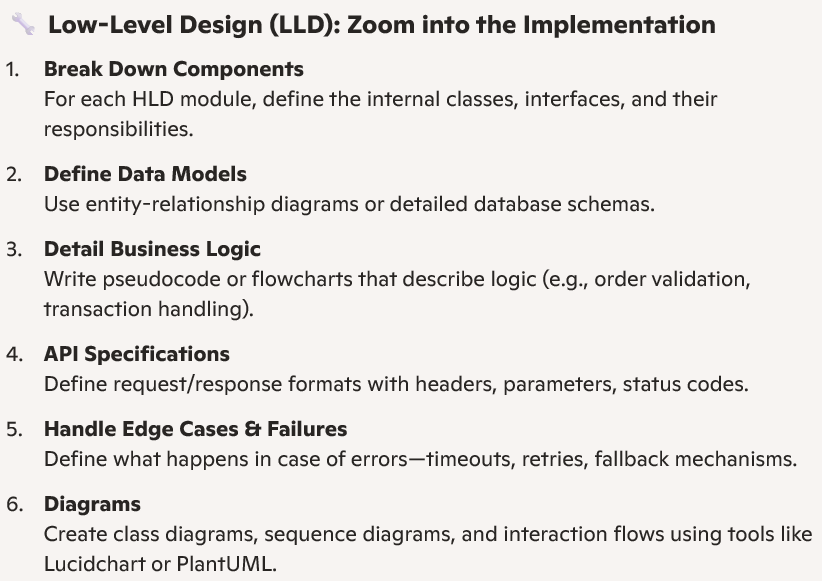




**HLD and LLD**

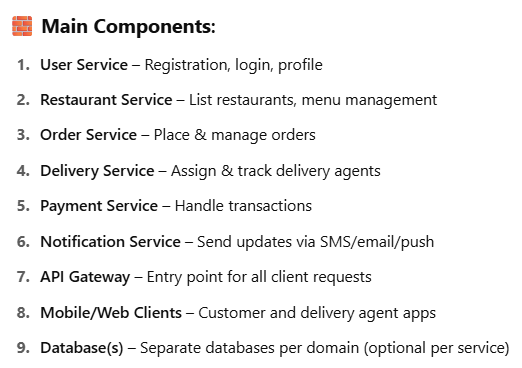
Creating HLD (High-Level Design) and LLD (Low-Level Design) for your project is like building a roadmap from the big picture down to the nuts and bolts. Here’s a step-by-step guide to help you design both effectively



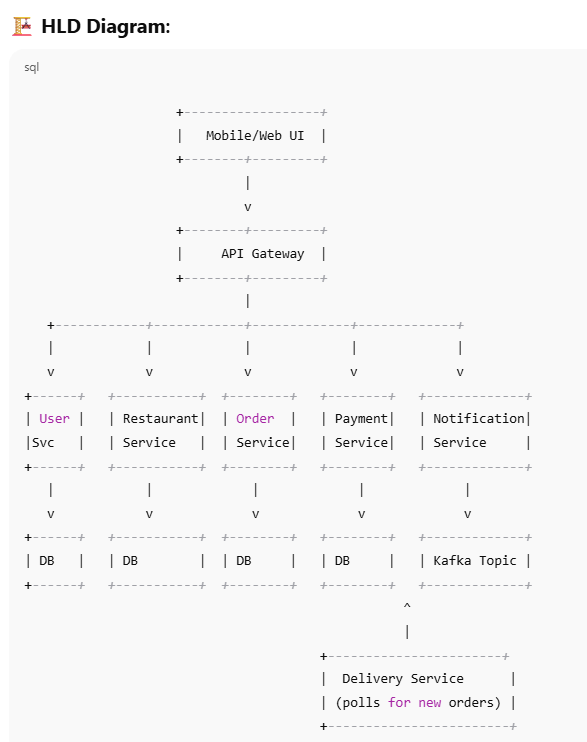


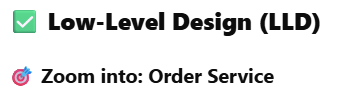
**Realtime Example: Online Food Delivery Platform (like Swiggy/Zomato)**

****

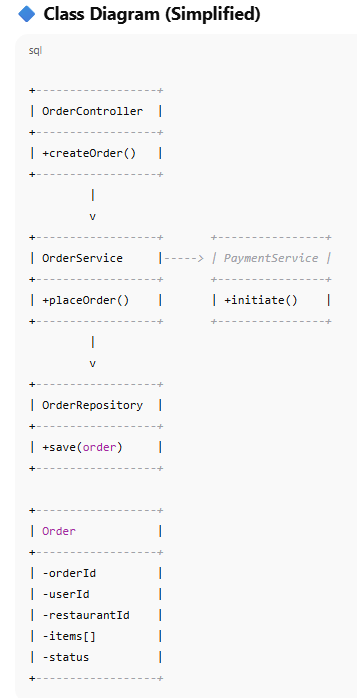
****

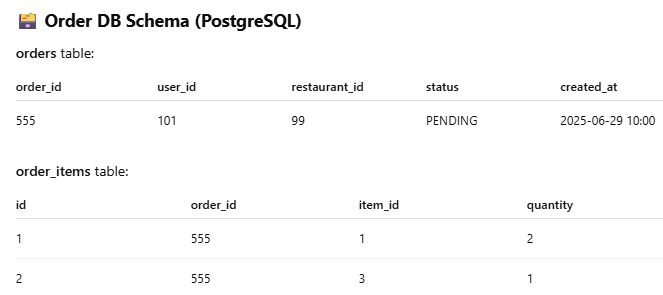
****

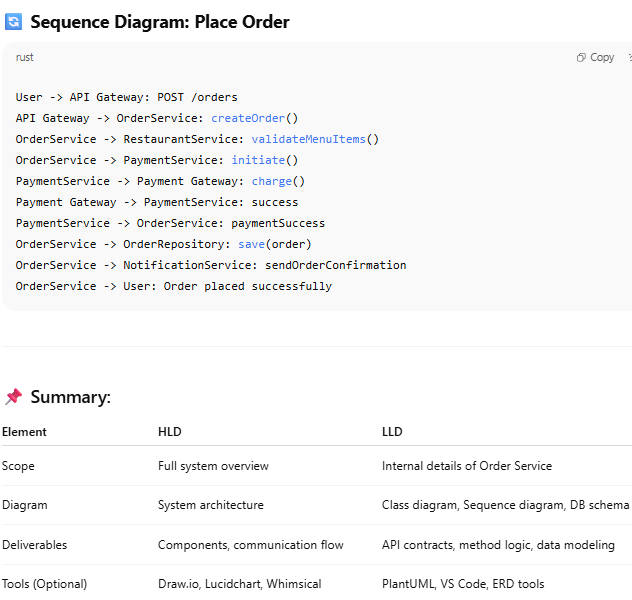
****

****

****

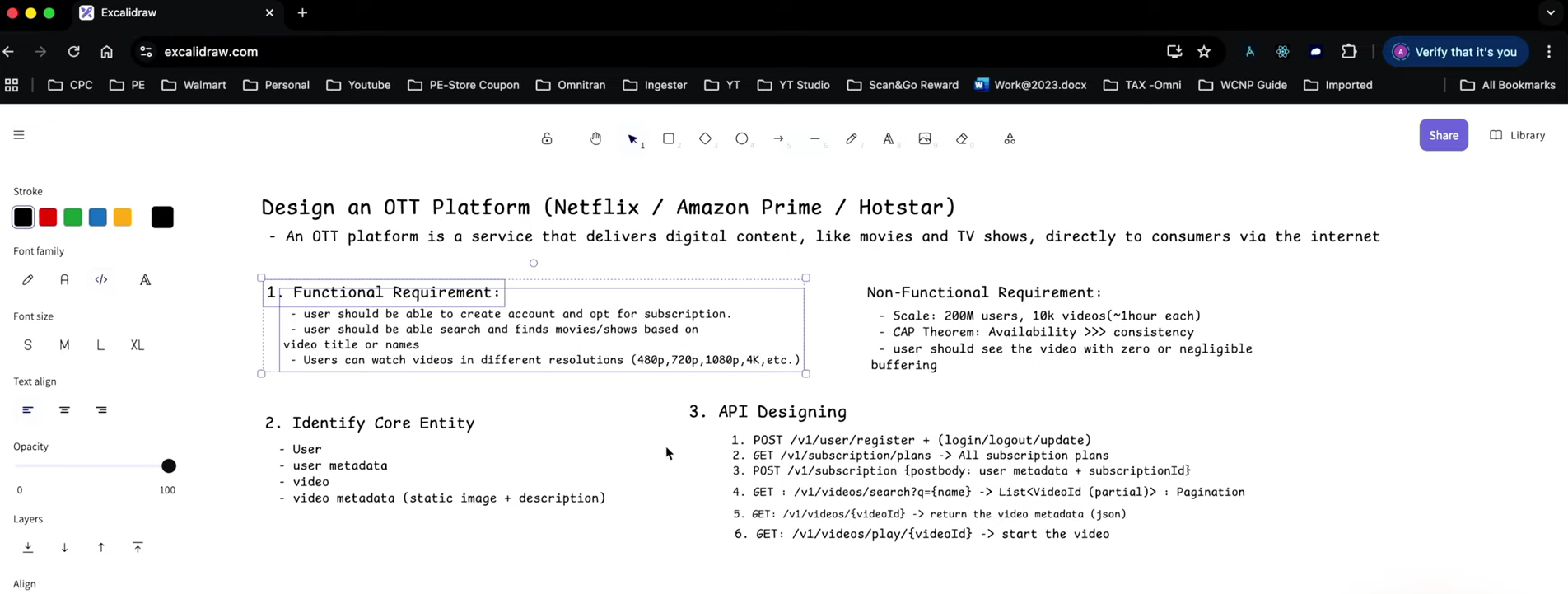
****

****

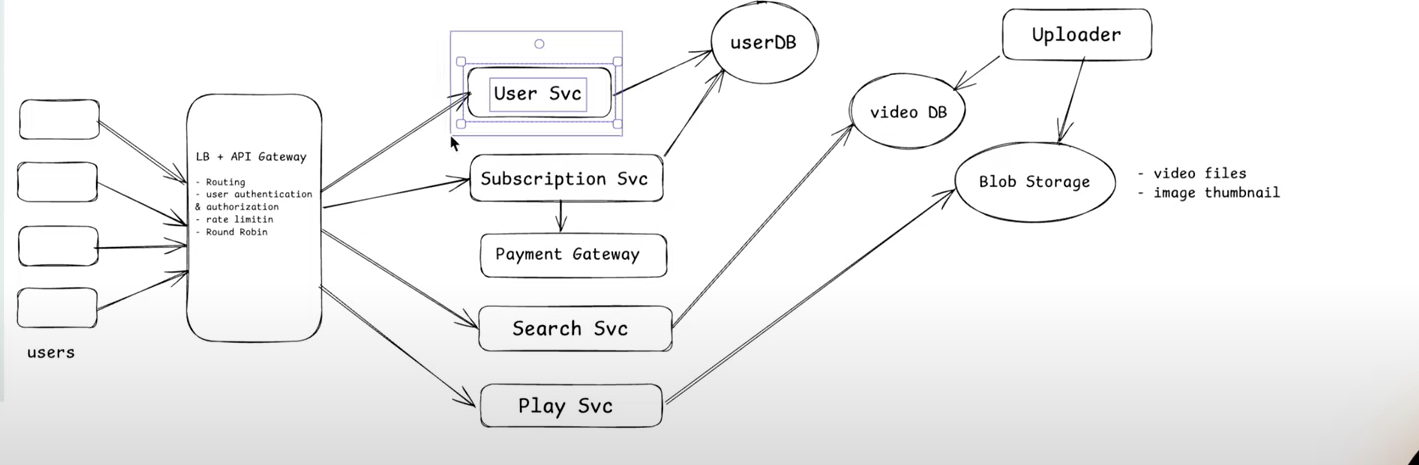
****

**System Design Netflix:**

**Functional and Non Functional Requirements**



**High level design:**



**Uploader ->** Will be handled by the Netflix system team

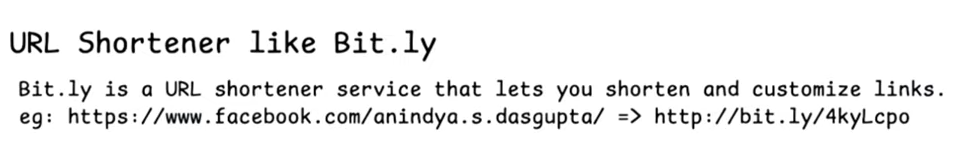
**Video DB ->** It will store the meta data about videos.

**Blob Storage ->** It is used to store the processed video.

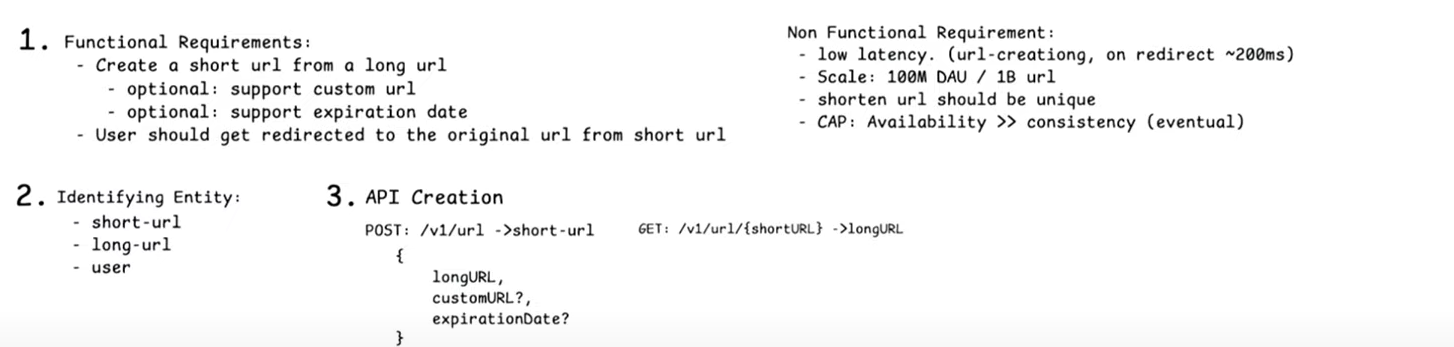
**LLD:**



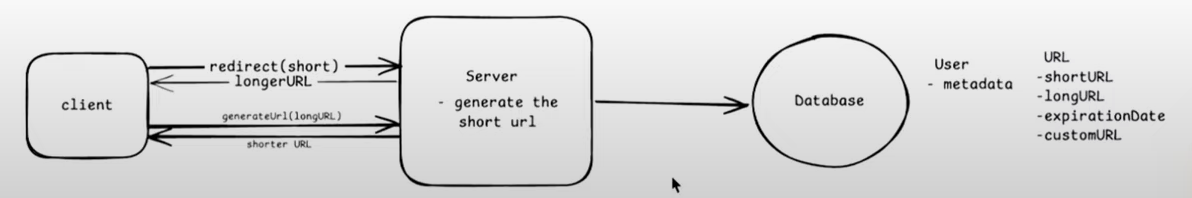
**System Design Tiny URl:**



**FR, NFR and API Design:**



**High Level Design:**

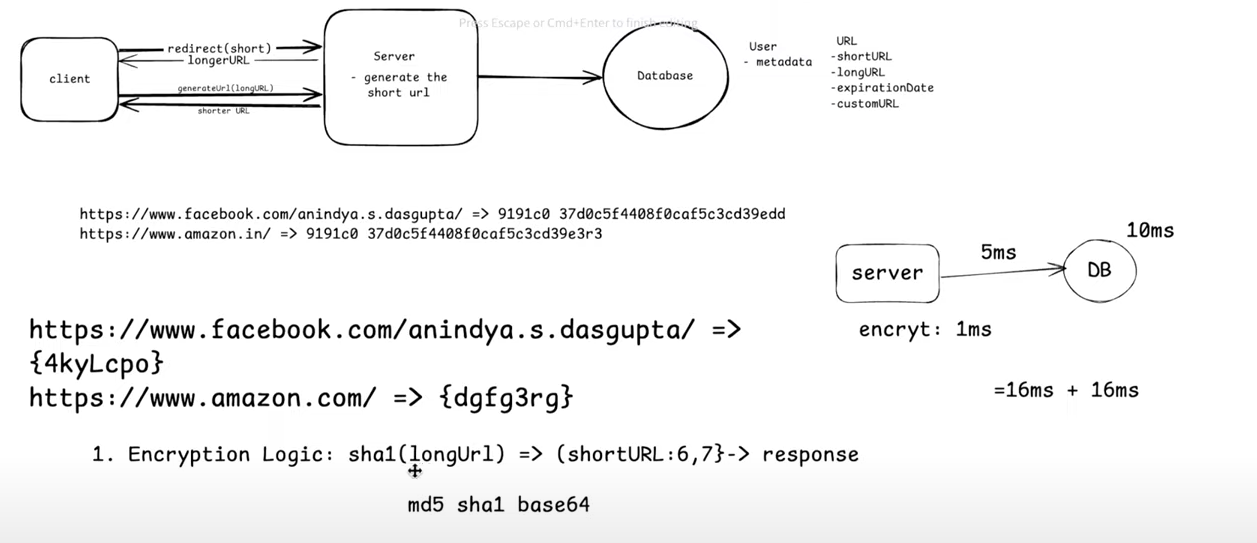
****

**Here two data bases will be created,**

**User -> Holding the user information**

**URL – For holding the shorter and longer url.**

**Low Level Design:**

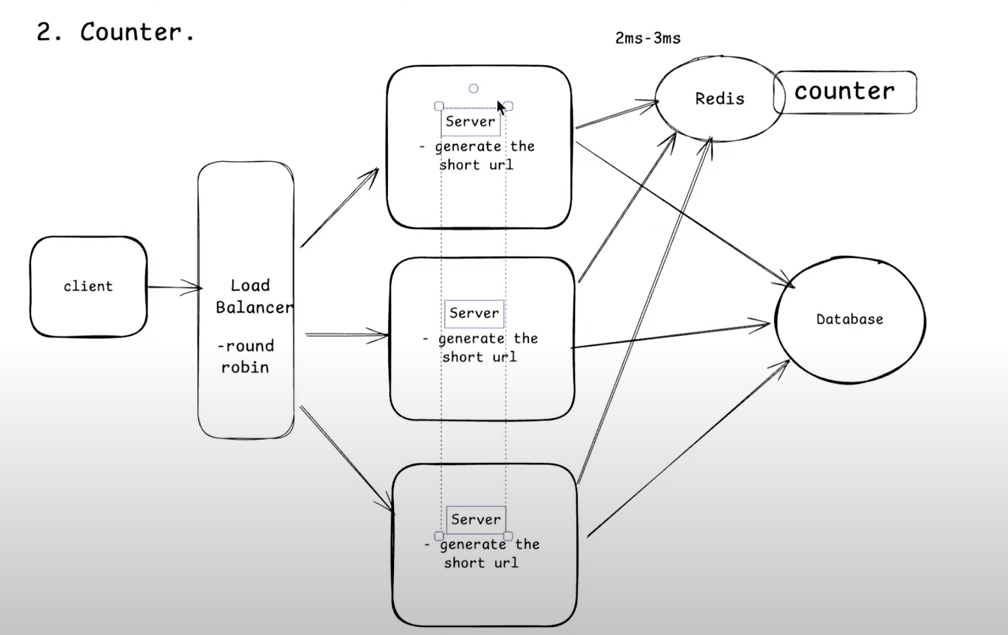
****

**First Approach Encryption Logic:**

**As a first approach we took the encryption logic. As you see hash generated for url is longer and it will consume the space so instead of returning the entire hash will return first six chars. Here there a chance of getting the same six digits for another url, hence first we need to check the existing url in the data base if not exists returns the existing one. Hence it’s a time consuming approach.**

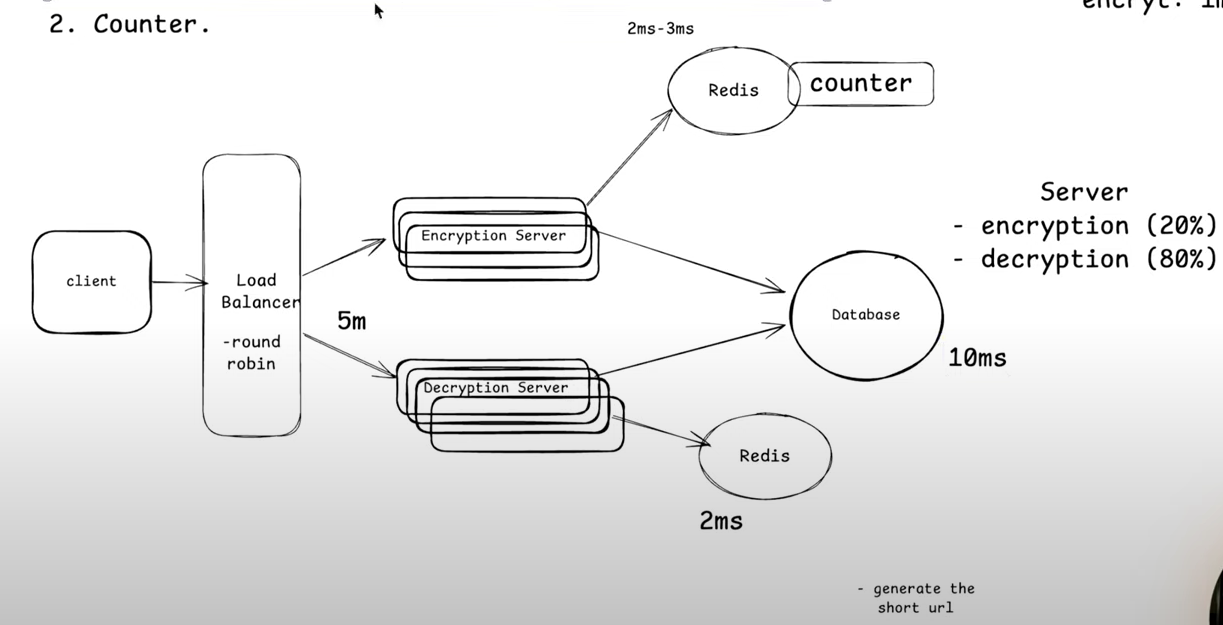
**In the above example its taking 16ms to check the new url existence in database its taking 10 ms and for encryption its taking 1ms and to save the url into database its taking 5ms.**

**Second Approach is Counter:**

**In the counter approach we will use the microservices as we are receiving billions of requests per second. To distribute the traffic equally among the services we will follow the horizontal scaling approach with load balancer as front door approach. In this approach we will maintain the counter at Redis cache level to generate the unique url in all the servers. Here the counter value will be cached and based on the exsting counter value in session next url will be generated. Here we are maintain the counter at redis cache instead of the server local cache because we don’t know load balancer send request to which server. If we maintain server level cache lets consider first two hits came to the server 1 then it will generate the url’s with count 0 and 1. In the next hit if load balancer send the request to send server again counter started from 0, so there is a chance of duplicate url’s. that’s why we are maintain global cache with redis.**

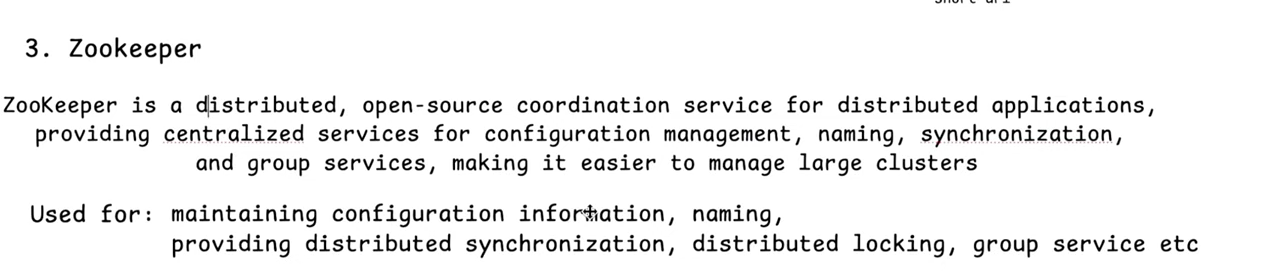
**Here the problem is redis cache, because if redis node is single point of failure. If we maintain the redis cluster with multiple redis servers after serving few requests if any of the redis server is down again the counter starts from 0 and there is a chance of duplicate data.**

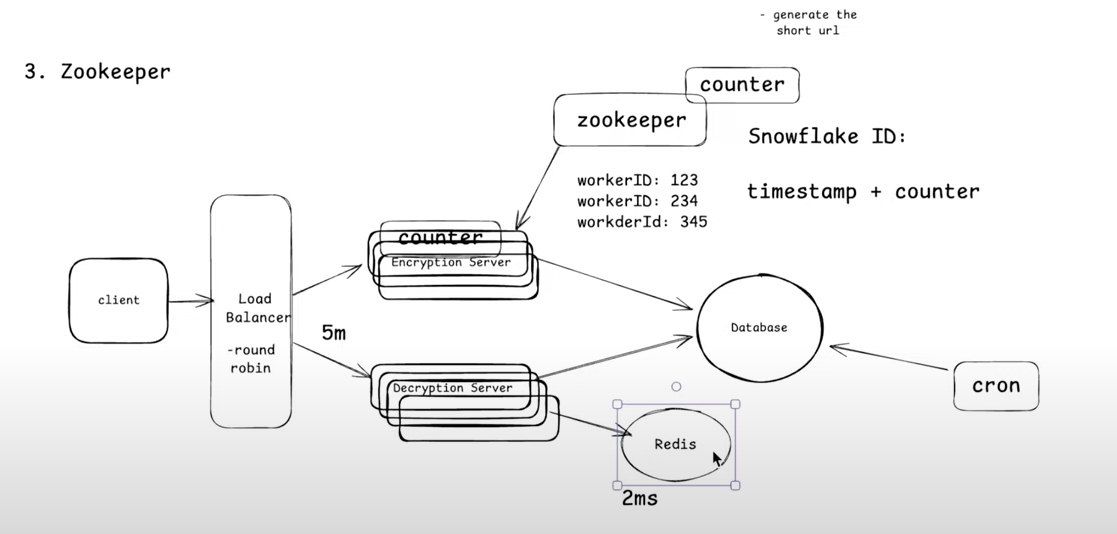
**In the counter another approach instead of keeping the encryption and decryption at on place will add the separate micro services for Encryption and Decryption. Here as encryption as less traffic we can scale 3 instance encryption server and 6 instances of Decryption server. To reduce the latency at decryption time we will maintain redis cluster at decryption as well.**

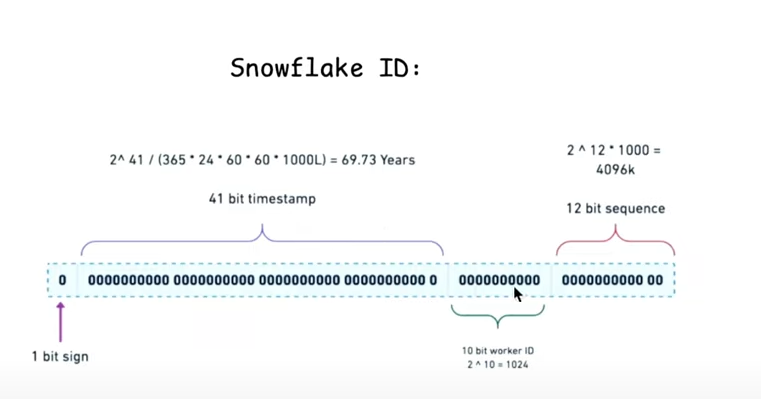
****

**In this approach also problem is redis cache, because if redis node is single point of failure. If we maintain the redis cluster with multiple redis servers after serving few requests if any of the redis server is down again the counter starts from 0 and there is a chance of duplicate data.**

**Final Approach with Zookeeper:**







In this approach we are replacing the redis cache cluster with Zookeeper. Here Zookeeper will manage the encryption and decryption cluster. In this case we will maintain the counter at Encryption server level. In this architecture zookeeper will generate the unique worker id(snow flake id) for each instance, with that unique id we can generate the unique tiny url for each request. Even in case anyone of the Encryption instance down the counter will be backed up at Zookeeper counter level. As we are not doing any interactions with zookeeper even if zookeeper fail we have backup counter at encryption server itself.

**Clean Up:**

As we are saving the billion of records in database and redis cache its not good to store the data for long time in database and redis cache. For this we should use cron job on daily basis to delete the expired records from database. In the redis cache at the configuration level we will use the TTL(Time To Live) for 30 days or 90 days to remove from cache.

For custom URL initially we check whether that url available at database if not will create new one.

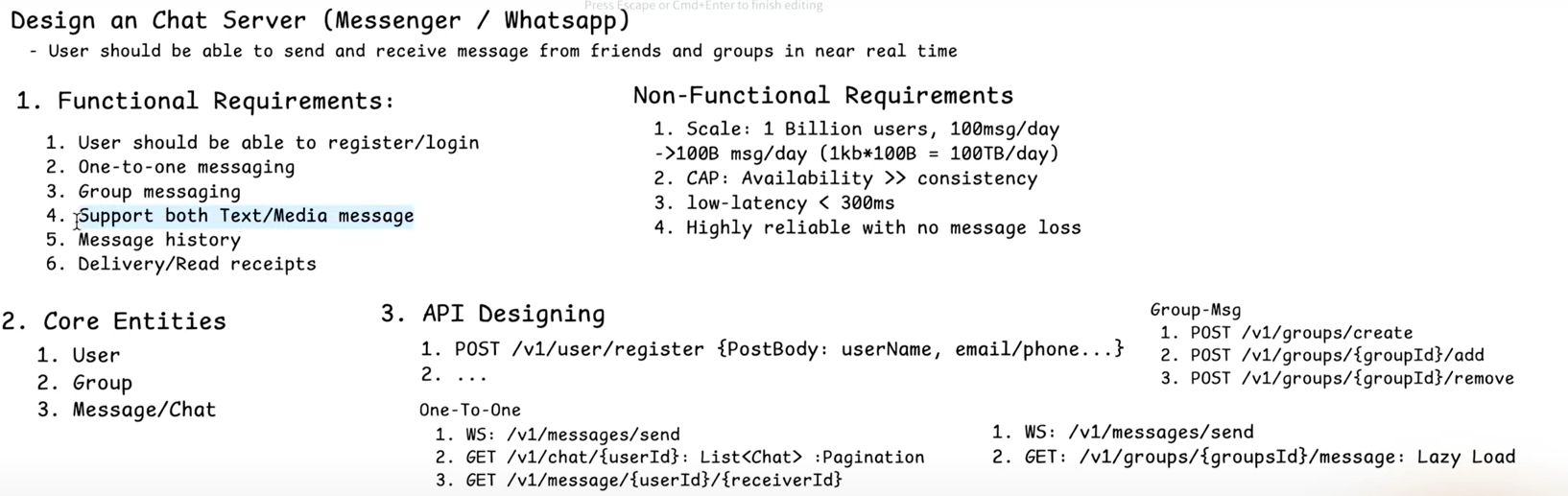
**For url redirection either will use 301 or 302.**

301 means permeant redirection means while redirecting the in local cache it will be stored. So it will reduce the number of calls and redirection time.

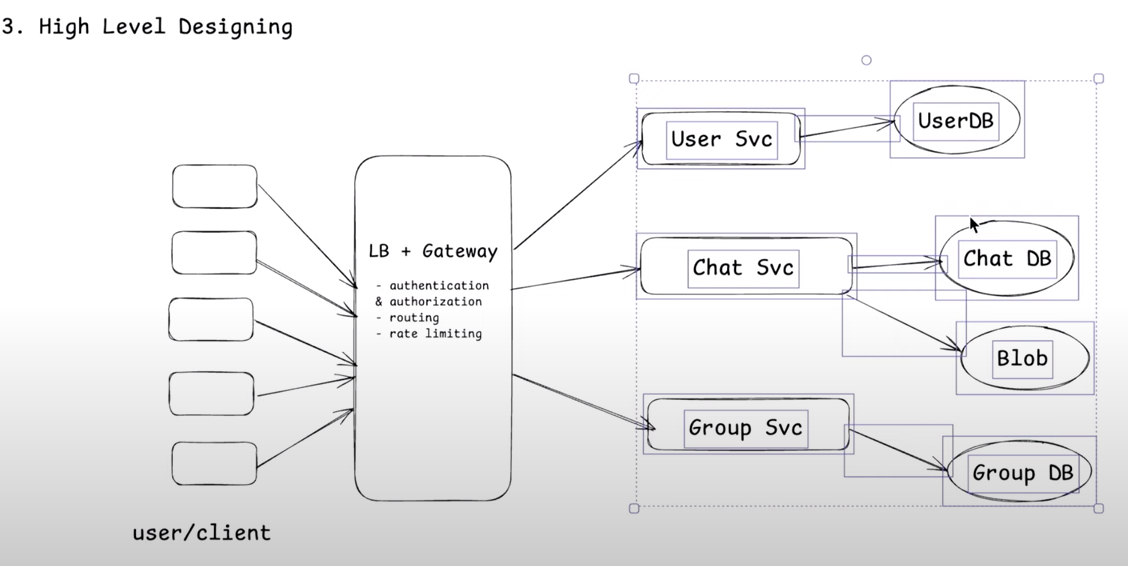
302 means every time it will hit the decryption server and fetch the corresponding long or short url form cache or database then it will be redirected. If we want to track the number hits in the form of metrics 302 will be best.

As we are not storing much data it good to go with either MySQL or Postgres Database. But create index on the short url for better performance.

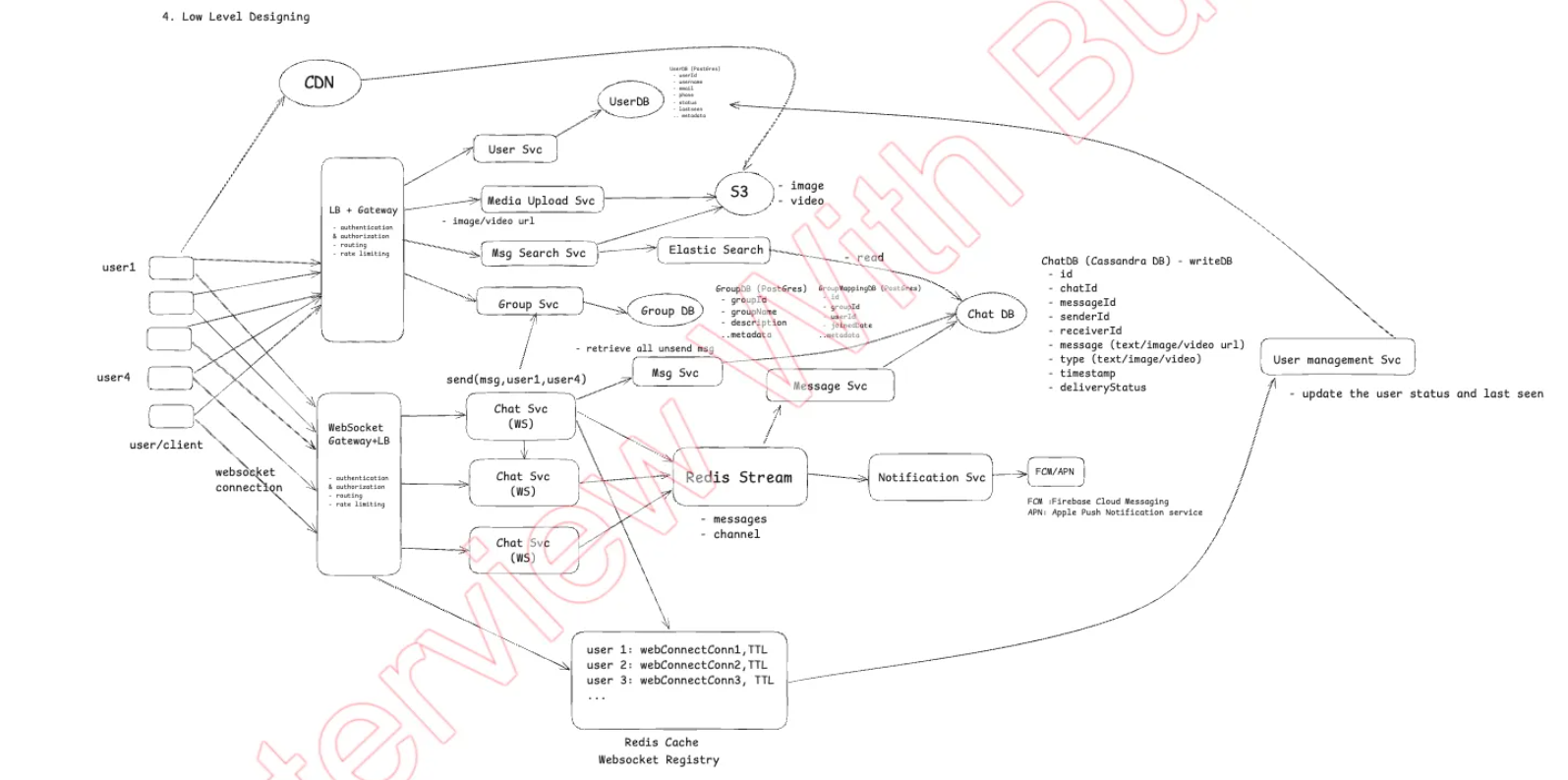
**System Design for Messenger:**

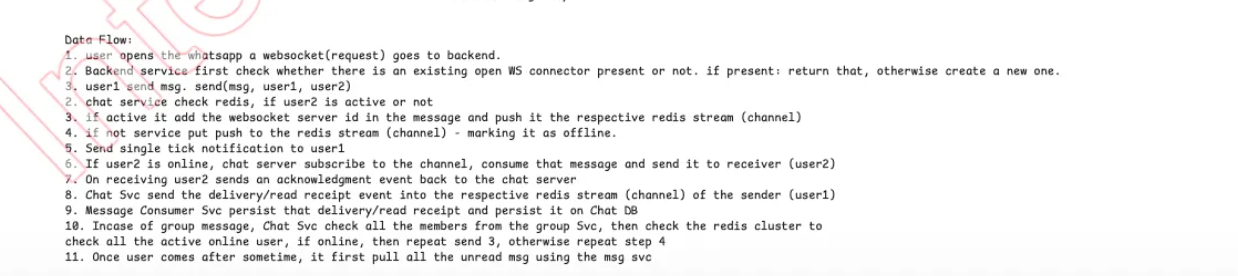


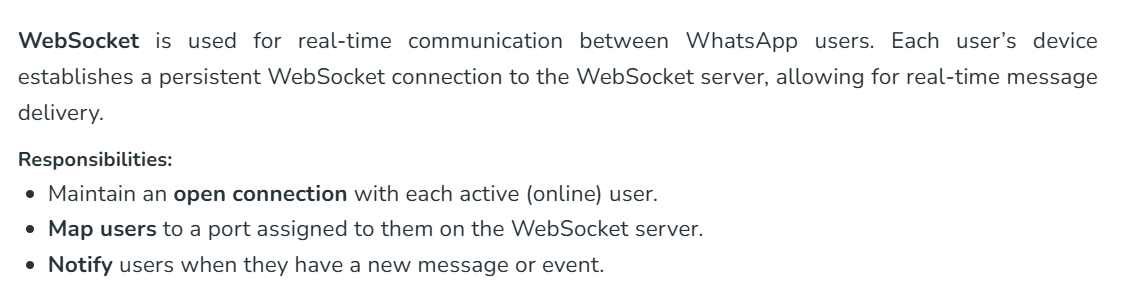
**High Level Design:**



Low Level Design:







**Uber/OLA System Design:**

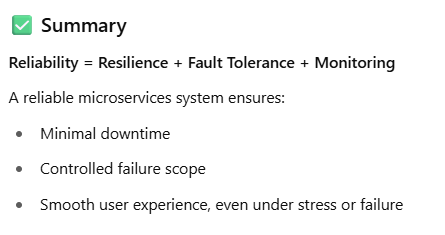
**Functional Requirements:**

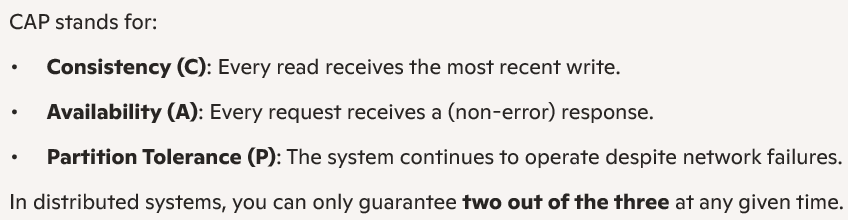
|  |  |
| --- | --- |
| **User** | **Driver** |
| Create Profile | Create Profile |
| Add start and end location | Show Availability |
| Show near by cabs/Bikes/Auto | Accept/Reject Trip |
| Show ETA and approximate price | Navigate to Pick Up Location |
| Book Cab | Start Ride |
| Make Payment | Navigate to Drop Up Location |
| Rate Driver | End Ride |
| See Past Trips | Payment |
| Raise complement | Rate Customer |
| Get current trip status | See Past Trips |
| Get particular past trip |  |

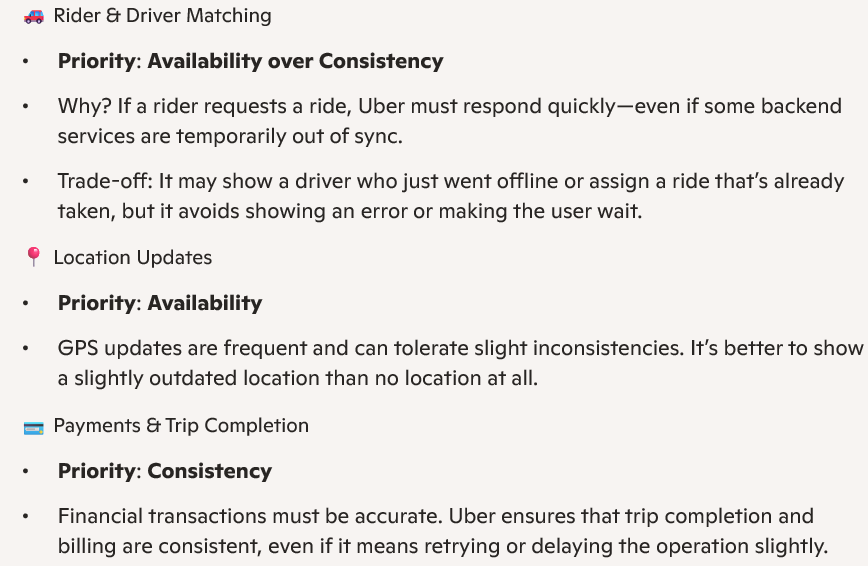
**Non Functional Requirements:**

* **Low Latency:** Latency refers to the time delay between a request being sent from one service and the response being received.
* **Highly Scalable:** It means the system can **handle increasing loads** (like users, data, or traffic) **efficiently** by **adding resources** (horizontally or vertically), **without performance degradation**.
* **Reliability:** Reliability in Microservices is all about ensuring that your distributed system continues to function correctly—even when individual components fail. Because microservices are loosely coupled and independently deployable, they introduce both flexibility and complexity. Here's how you can build reliability into them



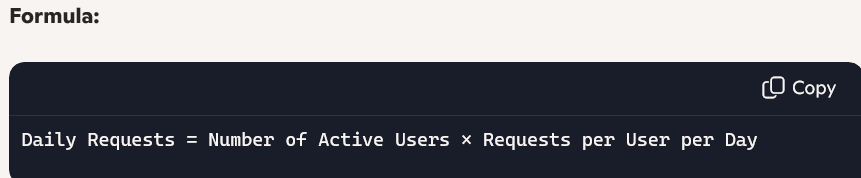


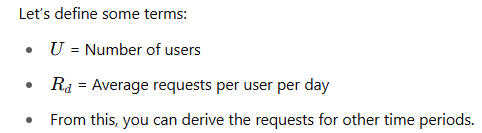
* **CAP Theorem:** 

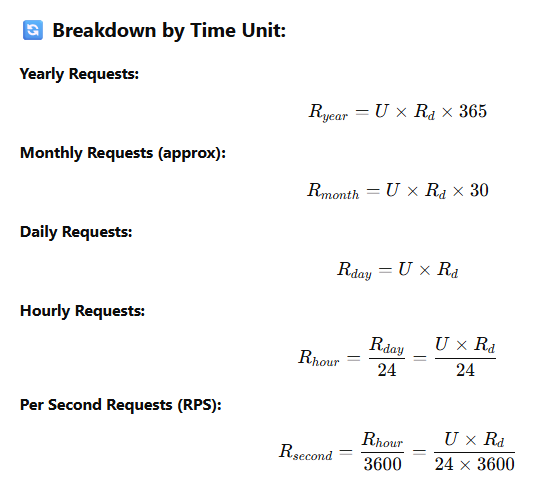


1. **Explain system design formula for calculating the requests received for year, Month, Hour and seconds?**

To estimate the number of requests your system might need to handle over different timeframes (like per year, month, hour, or second), you can use a scalable forecasting formula based on peak and average traffic patterns. Here's a practical breakdown







**Example:**

Active users: 100,000

Each makes 10 requests/day

Requests per year: 100,000 \* 10 \* 365 = 365,000,000

Requests per Month: 100,000 \* 10 \* 30 = 30,000,000

Requests per week: 100,000 \* 10 \* 7 = 70,00,000

Request per day: 100,000 \* 10 = 10,00,000

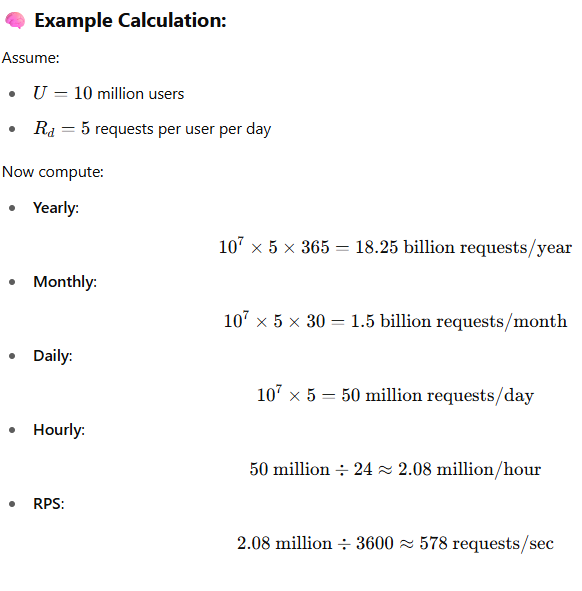
Requests per hour: 10,00,000/24 = 41,667 (Requests per day/24 Hours)

Requests per second: 41,667/3600 = 12 (Request per hour/3600 Seconds)

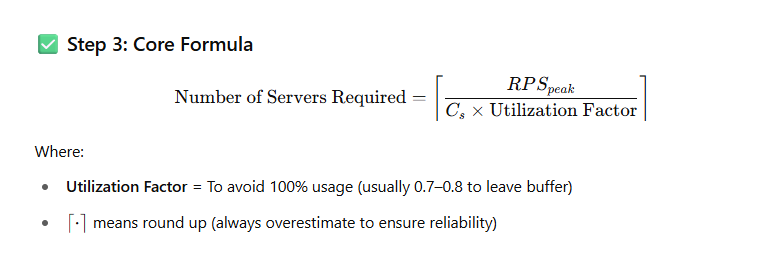
Here 24 Hours means 1 day in requests per hour calculation

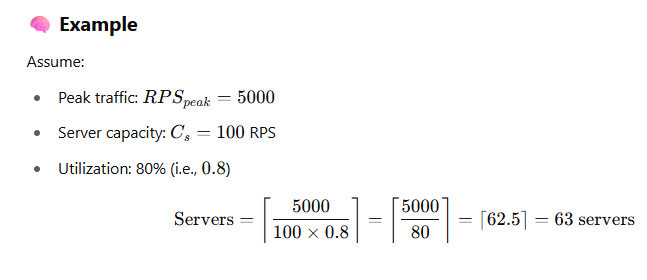
Here 3600 Seconds means 1 hour is equal to 60 minutes that is 60x60 seconds = **3600 seconds**

**Another Example:**



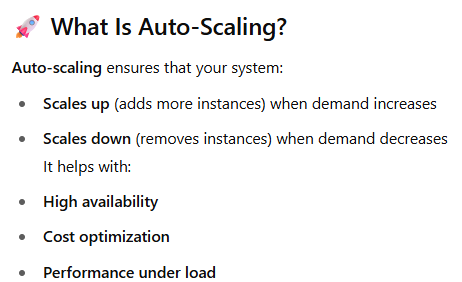
**Now calculating the Servers required based on the requests receiving:**

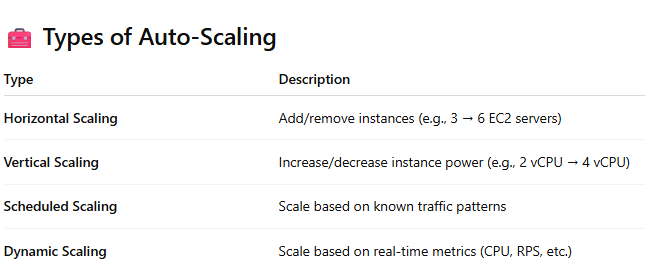


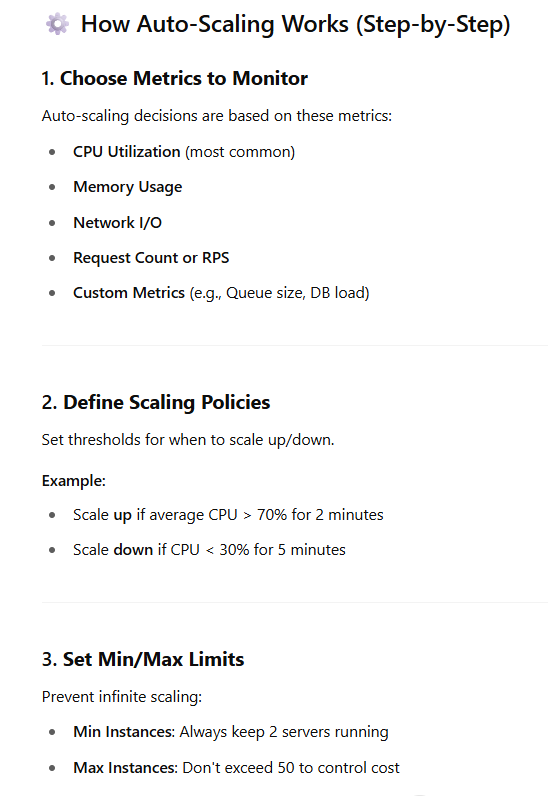


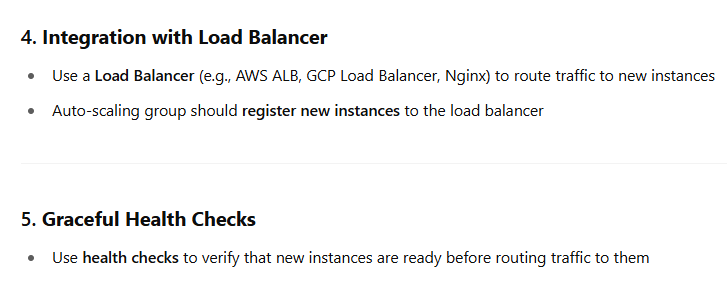
1. **Can you explain how autoscaling can be implemented?**

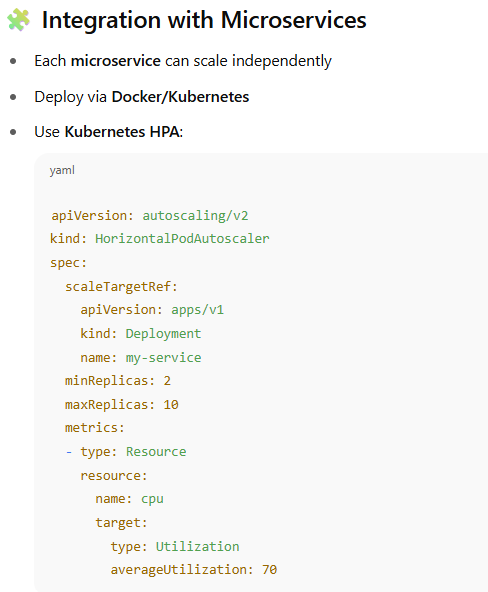
**Auto-scaling** is a key design pattern in **modern cloud architecture** that helps you automatically **add or remove servers** (compute resources) based on real-time load (e.g., traffic, CPU, memory).





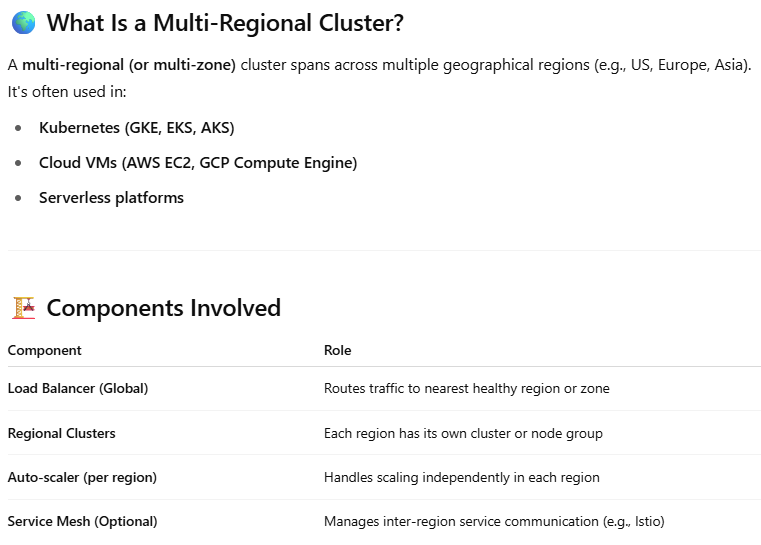


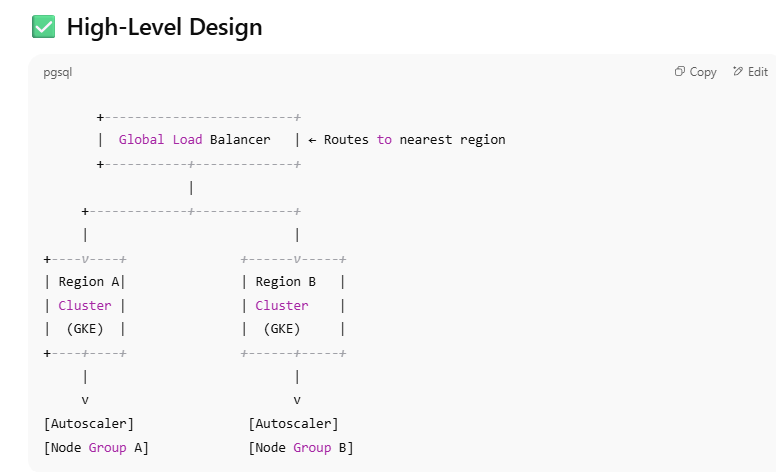


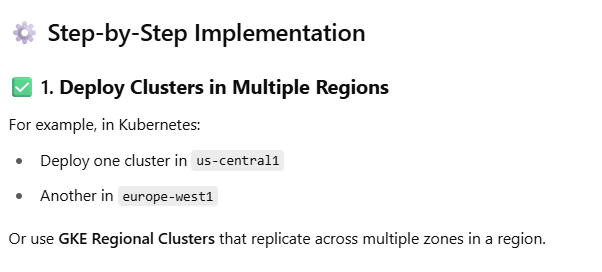


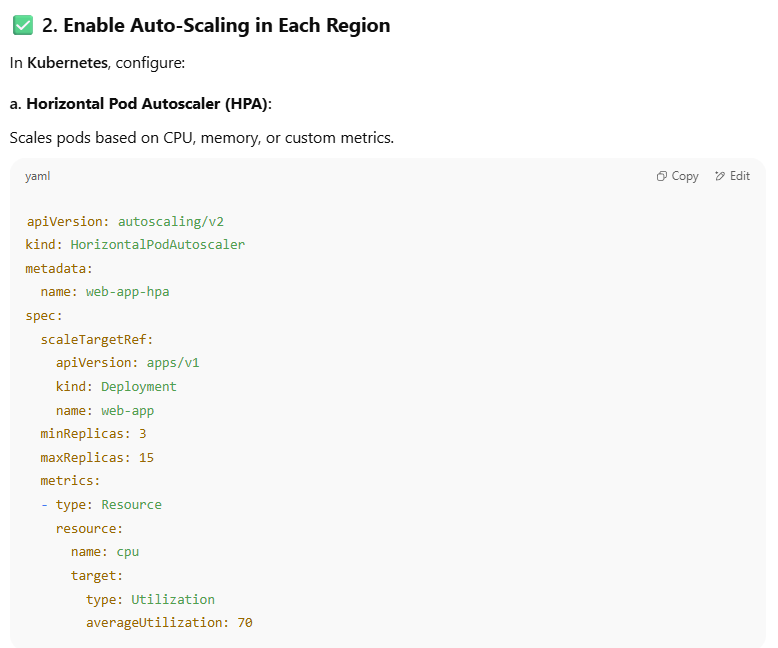
1. **Can you explain how autoscaling can be implemented in multi regional cluster?**

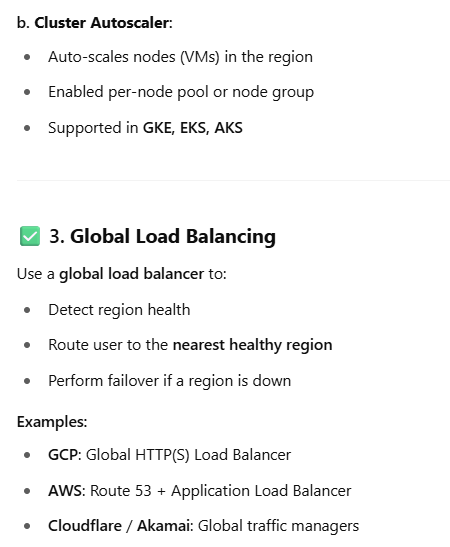
Implementing **auto-scaling in a multi-regional cluster** adds complexity but greatly improves **resilience, latency**, and **global availability**.



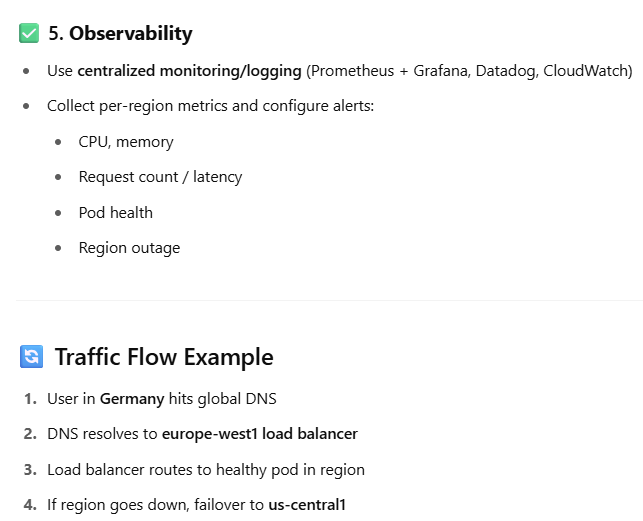


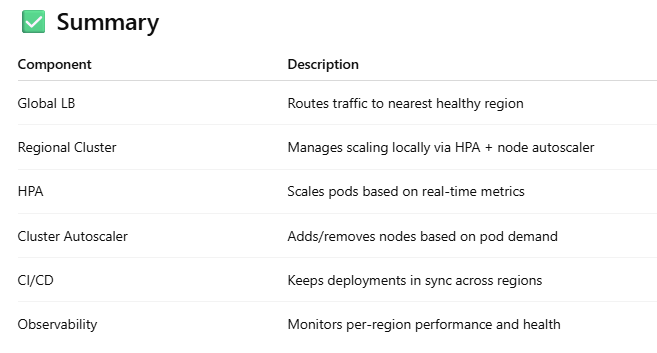




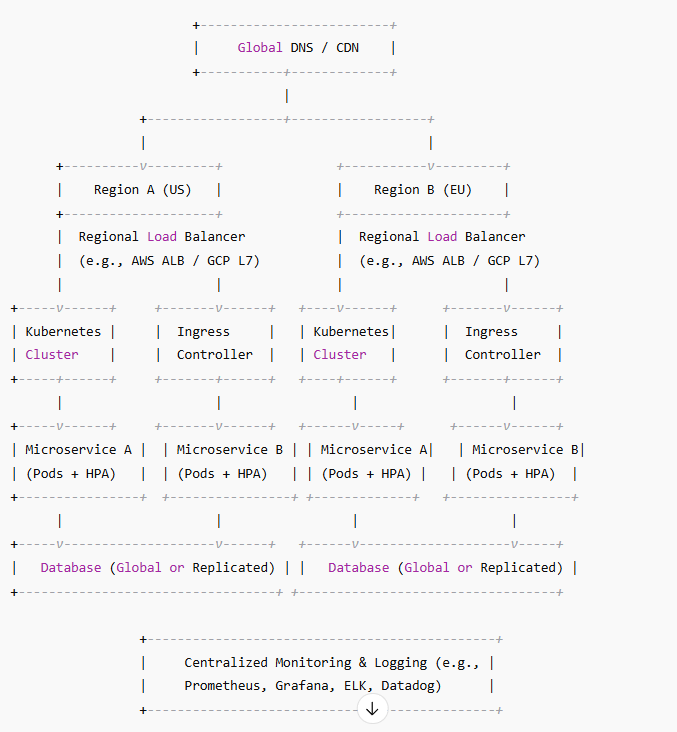


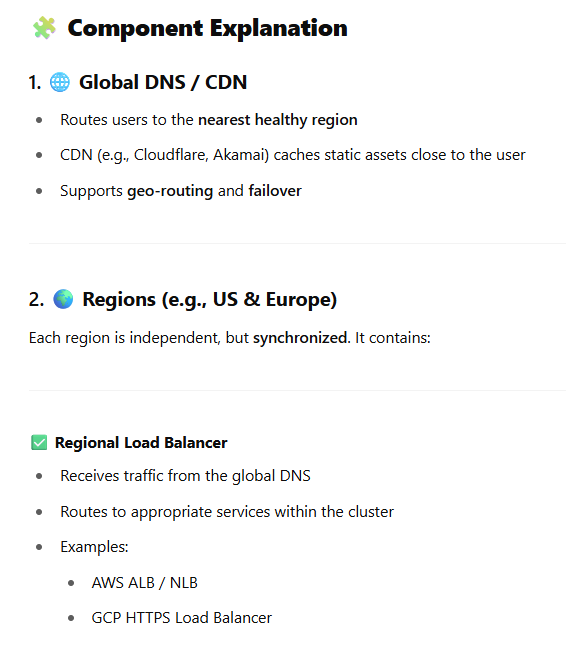


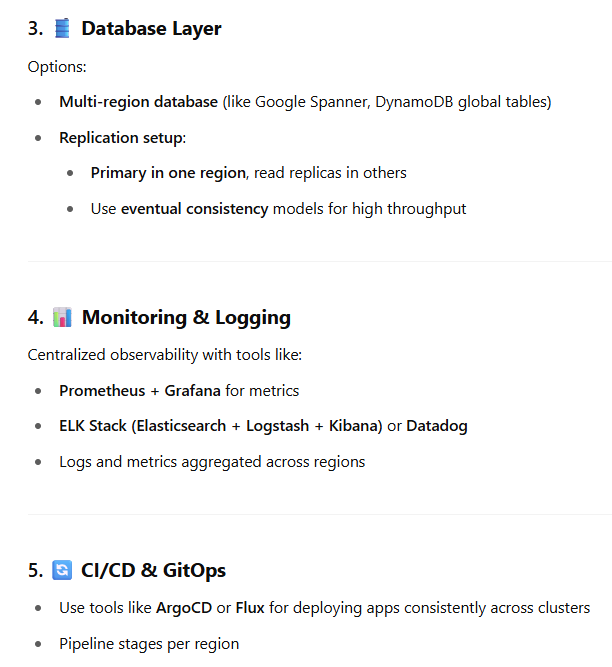
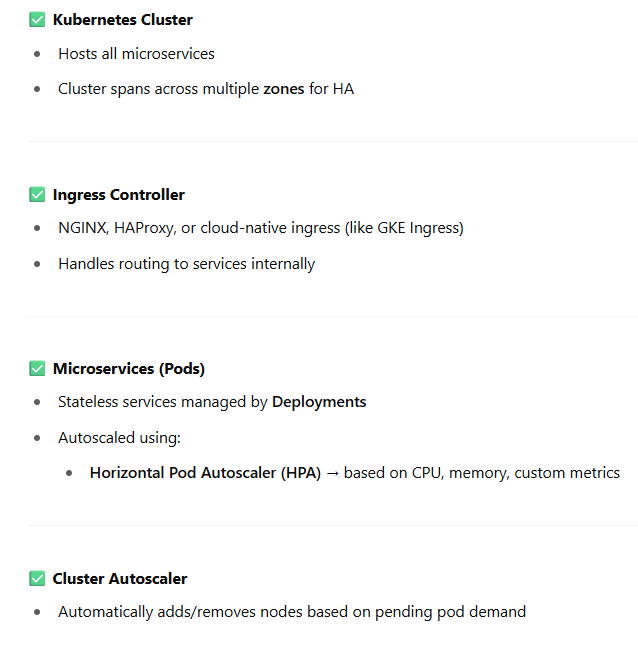


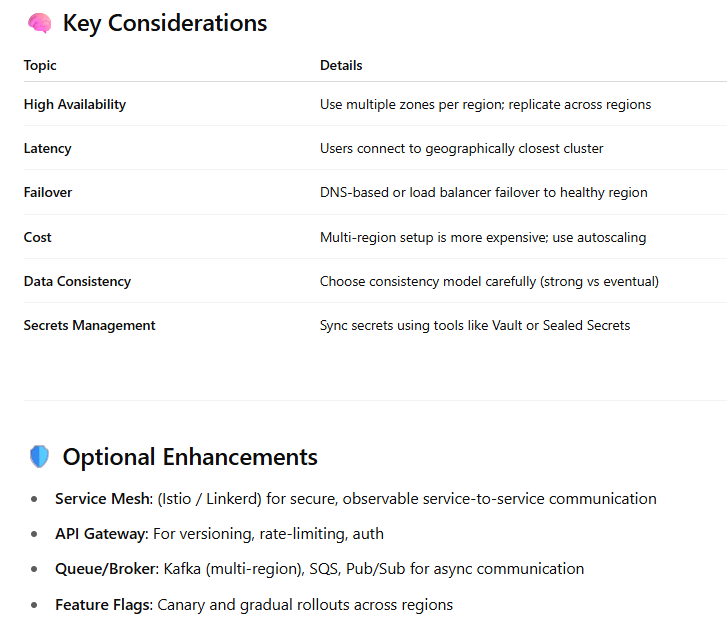


1. **Explain micro services multi regional Kubernetes architecture?**

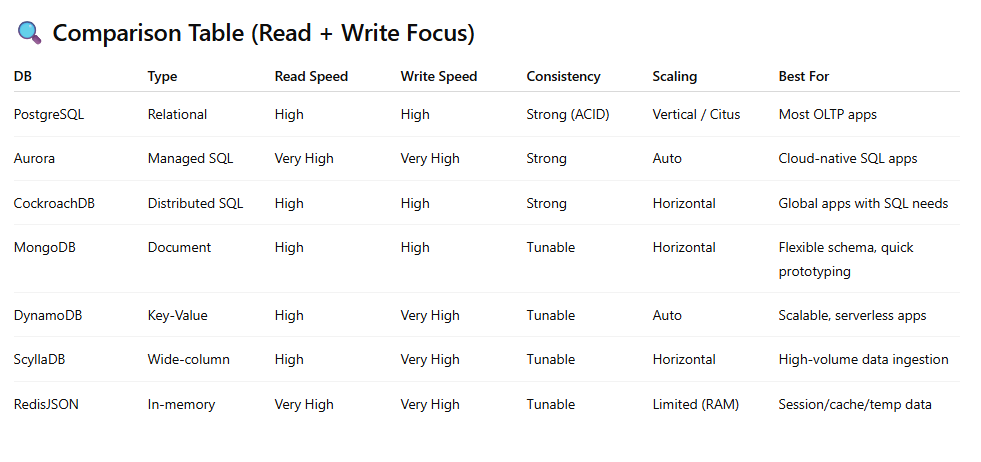


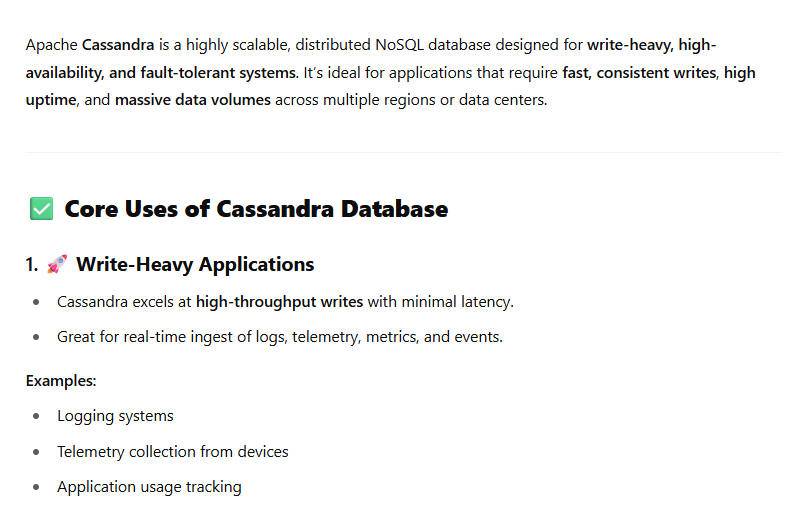




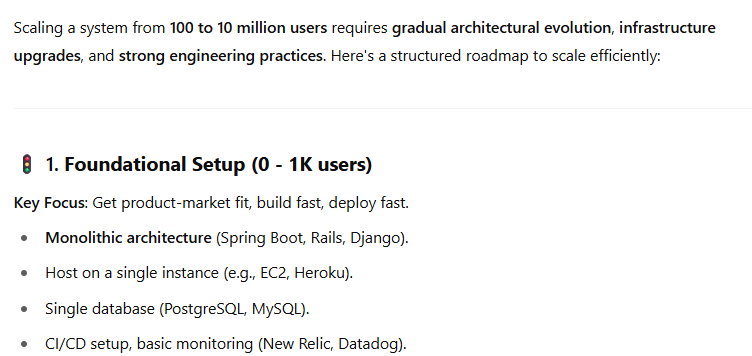


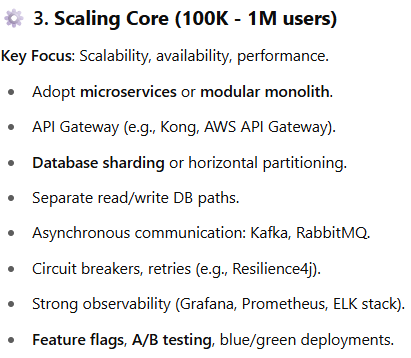
1. **Explain the better databases in micro services system design?**

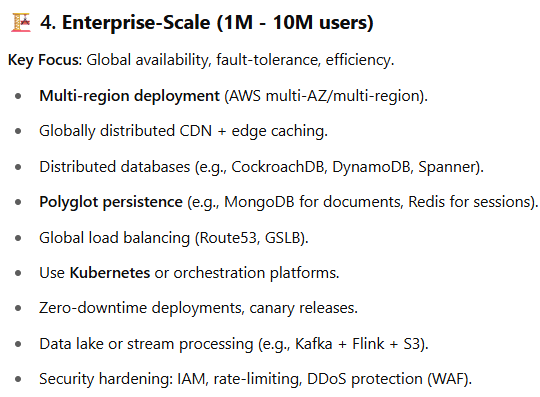




1. **How to scale the system from 100 users to 1 million users support?**

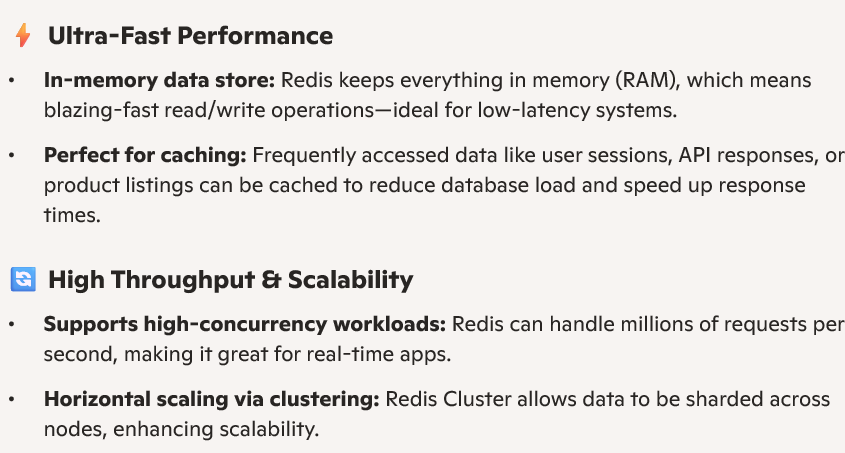


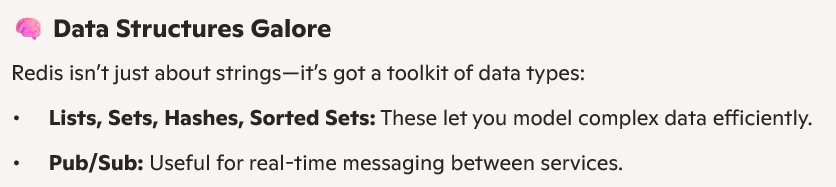


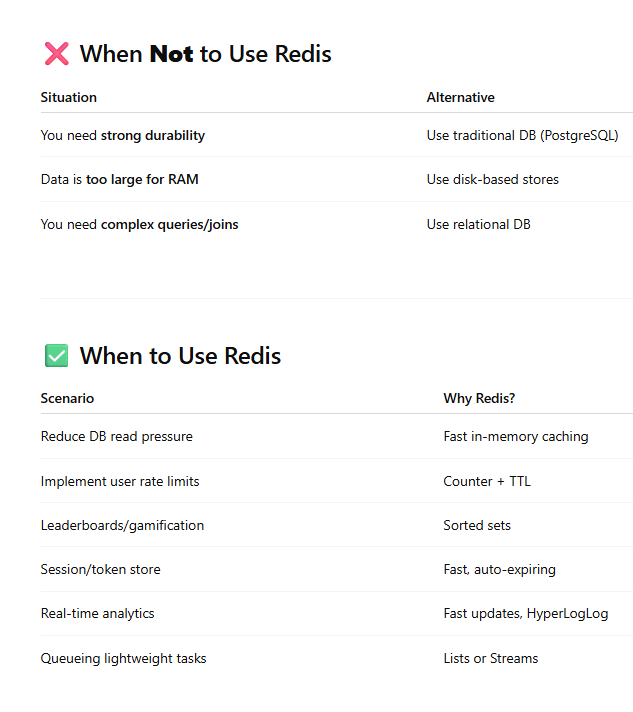


1. Why we use Redis in system design?

**Redis** (Remote Dictionary Server) is a **high-performance, in-memory data store** commonly used as a **cache, database, and message broker**. It’s chosen for its **speed, simplicity, and versatility**.

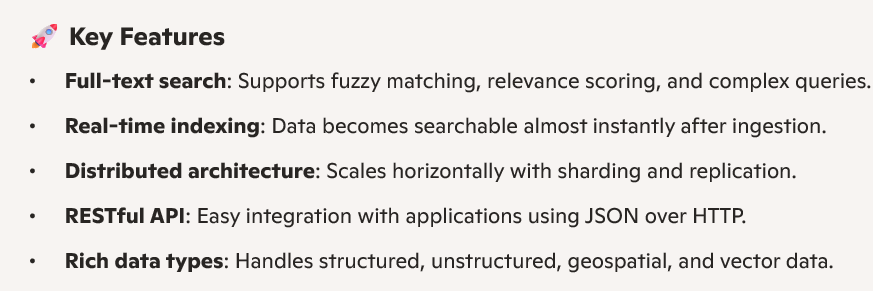


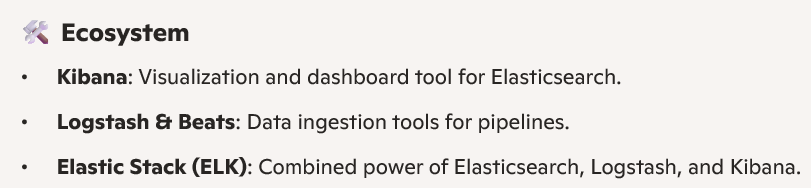




1. **Explain about Elastic Seach?**

**Elasticsearch** is a powerful, distributed search and analytics engine built on top of Apache Lucene. It’s designed for speed, scalability, and flexibility—making it a go-to choice for full-text search, log analytics, and real-time data exploration.





1. **Explain about Istio?**

**Istio** is an open-source **service mesh**—a dedicated infrastructure layer that manages communication between microservices in a distributed system. Think of it as the traffic controller, security guard, and observability dashboard for your microservices architecture.

